The Supervising Committee for Pranav Kumar
certifies that this is the approved version of the following thesis:

QoS and Efficiency for FaaS Platforms

Committee:

Mohit Tiwari, Supervisor

Mattan Erez

QoS and Efficiency for FaaS Platforms

by

Pranav Kumar

Thesis

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Engineering

The University of Texas at Austin

May 2019

Acknowledgments

I would like to thank Prof. Mohit Tiwari for giving me the opportunity to do research during my graduate studies. I would also like to thank Prateek Sahu for being an invaluable research partner for the initial part of this project and Huawei for generously funding this research.

I am also thankful to Willy and Chester for making my stay at UT a fun and knowledgeable one.

Pranav Kumar

The University of Texas at Austin
May 2019

QoS and Efficiency for FaaS Platforms

Pranav Kumar, M.S.E.
The University of Texas at Austin, 2019

Supervisor: Mohit Tiwari

Serverless computing or function-as-a-service (FaaS) provides a way to write applications composed of scalable and manageable independent tasks communicating seamlessly without developer involvement. Strict performance guarantees or service-level agreements (SLAs) provided by cloud vendors demand predictable performance of serverless applications. Performance predictability in a datacenter environment suffers due to contention for hardware resources. In this study, we evaluate the effects of contention on two FaaS platforms; AWS Lambda, an industry leader in serverless, and the open-source OpenFaaS serverless stack. We develop a complete set of microbenchmarks as well as end-to-end applications composed of multiple functions as a benchmark suite to facilitate our study.

We quantify baseline system costs of these applications across both stacks given traditional orchestration mechanisms in an isolated system. We also quan-

tify the same with co-located workloads in datacenter-like setting with Kubernetes orchestration. We show, via experiments, that significant performance slack exists at low to moderate loads and we can intelligently colocate workloads to maximize hardware utilization while still meeting QoS target latencies. Finally, we present a contention-aware static scheduling solution for FaaS platforms with predictable performance and compare it to static versions of baseline related works. We find that an intelligent FaaS orchestrator can be based along similar lines (similar hardware-level features) as a microservices one.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Application Development

Application development is generally split into two realms [3]: the frontend and the backend. The frontend is the part of the applications that users see and interact with, such as the GUI. The backend is the part of the application that the users do not see; this includes the server and the database where user's data and application state persists.

In the early days of the web, anyone who wanted to build a web application had to own the physical hardware required to setup and run a server, which was a cumbersome and an expensive undertaking. Then came the cloud, on which we could rent fixed number of servers or server space remotely. Developers and companies who rented these fixed units of server generally over-purchased to ensure they could handle periodic or ephemeral spikes in traffic.

1.2 Serverless Computing

Serverless computing is a method of providing backend services on an as-used basis. A serverless provider allows users to write and deploy code without the worry of the exact underlying infrastructure setup. Any company or individual that gets backend services from a serverless vendor are charged based on their computation and resource usage. They also do not have to reserve and pay for a fixed amount of bandwidth or number of servers, as serverless services are usually auto-scaling. Although called serverless, physical servers are still used however developers do not need to be aware of them.

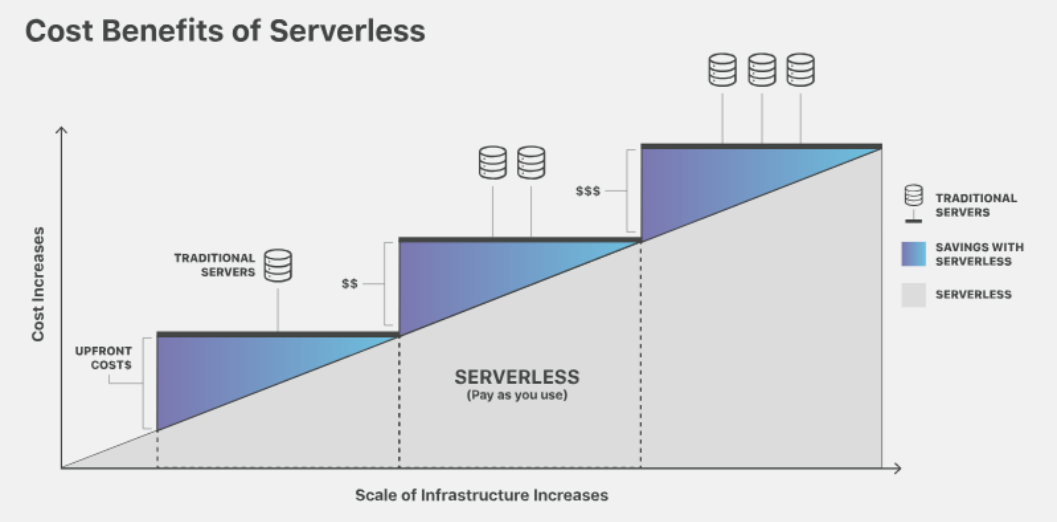

Figure 1.1: Cost benefits of serverless

Serverless computing allows developers to purchase backend services on a flexible pay-as-you-go basis, meaning that developers only have to pay for the services they use.

1.2.1 Advantages

- Lower costs: 'pay-as-you-go' model is usually very cost-effective
- Scalability: Serverless vendor handles all of the scaling on demand
- Simplicity: developers have to only worry about writing the backend code and none of the server management
- Smaller time-to-market (TTM): Serverless significantly cuts the time to deployment

1.3 Function-as-a-service

Function-as-a-Service(FaaS) is a new paradigm of micro-services (in some sense, a serverless way) in software architecture and development cycle used by most tech companies like Netflix, AirBnB etc. The usability of micro-services comes from the distributed nature of services deployment which enables easy scale-up and maintenance of specific parts without tearing down the entire stack and potentially avoiding large downtime.

1.4 Functions

Functions are a layer of abstraction over containerized services, where the developer is oblivious to underlying hardware. The software stack, which is often provided by Public and Private Cloud vendors like Amazon[4], Azure, Pivotal etc, manages the deployment and runtime orchestration of these functions. But as such services grow their public cloud usage, they need certain guarantees provided by vendors like Amazon Web Services in terms of reliability, turn-around time and latency.

Functions tend to be event driven small units of services which are spun up when a trigger arrives and are usually short lived (few milli-seconds to tens of minutes). These functions leverage native Linux containers to easily create and destroy instances and make use of container orchestrator like Docker Swarm[5] or Kubernetes [6]to scale up according to request demand. To meet the Service Level Agreements, the vendors tend to over-provision resources for these containers and use custom orchestrators to schedule them them. The software stack is often not optimized for hardware utilization but rather towards general usability across hardware. This leads to huge loss of compute cycles and eventually loss of performance because they do not leverage hardware primitives like performance statistics etc.

Most of the cloud pricing model, including FaaS , are driven by resource allocated and time-slice used for the resource. Resources like VM instances are billed per hour or Storage with S3 is with per 1GB. But AWS Lambda a granularity of 100ms of use. With FaaS the time slice becomes a significant factor since functions are designed to be ephemeral and do not run for too long. Cloud vendors would want to charge cheaper and utilization of resources during function's run time plays a huge role in determination of this pricing model. Consumers pay more for various guarantees for their function which essentially translates to more exclusive resource allocation to meet the timing demands.

With more and more applications being moved to FaaS platforms, vendors want to maximize their utilization as well meeting the provided SLA agreements to offer a competitive and lower price for their services. Among these FaaS applications, the most common are ones with small and latency-critical kernels. FaaS workloads, by essence, are small and loosely-coupled ephemeral functions which are executed in a lightweight container runtime environment based on a trigger and then killed

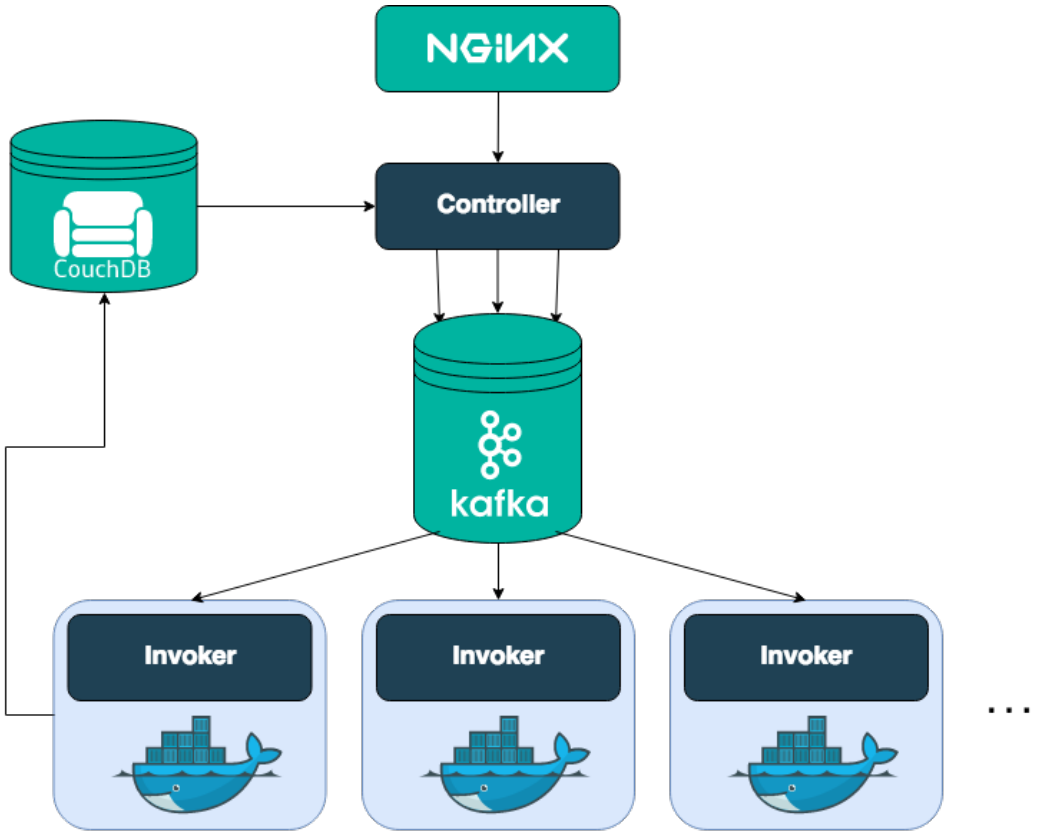

Figure 1.2: A general FaaS architecture [1] showing different event triggers (various linked storage and web service apps), a controller and multiple function invokers running on Docker.

soon after. A general architecture diagram for FaaS platform is shown in Figure 1.2. Batch jobs e.g. streaming, on the other hand, are long running tasks which do not usually act upon some trigger, rather are lasting background jobs. We refer to them as and tasks respectively. The long-running tasks are essential for an efficient utilization of a datacenter's resources. The use of FaaS also increases the number of RPC which can contribute to added latency in a system. this project is a new approach towards providing efficiency to cloud providers by characterizing of FaaS workloads. We ultimately work towards a scheduling methodology which provides latency guarantees while improving overall resource utilization of nodes.

1.5 Microservice vs Function-as-a-Service

Digital transformation from monolithic servers to microservices to serverless is driven by the need for greater agility and scalability [7]. Microservices architecture emerged as a key method of providing application development teams with flexibility and other benefits, such as the ability to deliver applications fast using infrastructure-as-a-service (IaaS) and platform-as-a-service (PaaS). The concept of this idea was to break the monolithic applications into smaller services each with its own business logic. These smaller services could be independently coded (in different programming languages), maintained and scaled according to the load. In a microservice, each service runs in its own container. Microservices involve source code management, a build server, code repository, image repository, cluster manager, container scheduler, dynamic service discovery, software load balancer and a cloud load balancer.

Serverless makes the unit of work even smaller. It takes a step further to break down an application to the granularilty level of small functions and events as

described in the previous section. FaaS also improves the shortcoming of PaaS model i.e. scaling and friction between development and operations. The key difference is that, in case of a function, the container is created and destroyed by algorithms used in FaaS platforms and the DevOps team have no control over that.

Though, there will be always space for both microservices and FaaS to co-exist because there are certain things which we cant do with functions at all. For example, an API/Microservice will always be to respond faster since it can keep connections to databases and other things open and ready. Moreover, one more thing to note here is that by grouping a bundle of functions together behind an API gateway, weve created a microservice. This shows that both of these can co-exist.

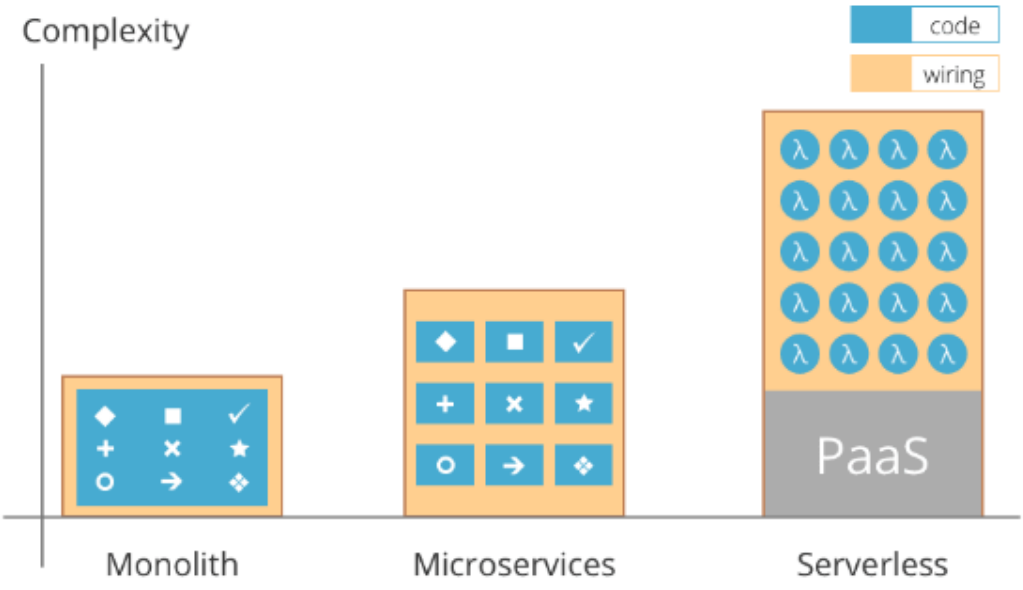

Figure 1.3: Paradigm transformation from monolith server to a microservices architecture to a serverless approach

1.5.1 Orchestration Techniques

Functions execute at a much more fine-granular level than microservices. Function invocations are handled by an orchestrator by spinning up and down containers when necessary whereas a microservice architecture is ready all the time. Both of them can handle multiple concurrent requests by auto-scaling. Thus, orchestration techniques for FaaS workloads have to be active on a much more fine-granular level than for a microservice architecture.

A FaaS orchestrator also has to take cognizance of cold-starts. Since the container is being spun up when required and down when it has been idle for a while, an intelligent orchestrator should ideally predict the up times and down times and take preemptive action based on the same. Since the FaaS software stack is not optimized for hardware utilization or latency or performance but general usability, it is the job of the orchestrator to still try to grasp as much performance out of the system as possible while maintaining high hardware utilization. Ideally, datacenters want to achieve close to 100% hardware utilization to minimize costs.

Thus, FaaS architecture being ephemeral, event-driven, asynchronous and low-latency in comparison to a microservice architecture might require exposing of different/additional hardware features to the scheduler for efficient orchestration. Intuitively, we look into high-level metrics like CPU usage, memory working set, etc. These metrics are already exposed by Docker and can be read from the Docker stats per sampling interval. Ideally, we want a better metric which gives us more low-level insights into an apps behavior. This metric would help us quantify the intrusiveness of a program when colocated with other workloads and when a particular colocation use-case is good vs bad. We make use of performance counters to identify suitable candidate metrics. The performance monitoring unit (PMU) based metrics are as

follows.

- IPC: instructions/cycle, **compute intensity**
- Cache MPKI: misses/kilo-instructions, **memory pressure**
- Branch MPKI: mispredicts/kilo-instructions, **pipeline front-end stalls**
- Page faults: long latency memory trips

Since we would only be exploring static orchestration techniques, we would later go on to create static versions of related work, which also use similar hardware-level metrics for orchestration. We would be starting off with a combination of the high-level and PMU-based metrics and based on feedback from the static versions of baseline related work, we will iterate on our own design. The motivation for this work is to know whether the design of an intelligent FaaS orchestrator would be along similar lines as that of a microservices one or would we need additional function-specific information to colocate effectively.

Chapter 2

Background

In this section, we briefly summarize background most pertinent to this project , including a description of our target workloads, the goals of performance and QoS techniques in the context of micro-services.

2.1 Cloud Workloads

Prior publications [8] classify cloud workloads into three categories. The first includes tasks that are not user-facing, for which throughput is the major concern and that can be freely scheduled in the background when resources become available. The second class includes short latency-critical user-facing tasks such as responding to web search requests and content caching. These workloads are characterized by short deadlines in the order of tens of milliseconds. The third major class of workloads corresponds to offloading of work from user devices to the cloud like online video processing, online stream data analysis and detection/recognition tasks. These are user-facing as well as latency-critical while having longer running times simultaneously. FaaS workloads specifically are growing in the space of the third

class of workloads with applications like image processing using metadata from a database/persistent storage, machine learning training and inference jobs, image/video conversion in the cloud, mobile back-end, chat bots, authentication etc. All of these have significant latencies, and are user-facing and performance-critical. We specifically target this third kind of workloads as our tasks and the first kind of batch workloads as our tasks. This is a reasonable assumption as datacenters tend to utilize resources effectively by running batch jobs in the background.

The performance of all these kinds of workloads for cloud providers are expressed both in terms of throughput and in terms of strict latency constraints. A very small percentage of total tasks are usually allowed to violate these constraints and these are what make up the SLA in contracts.

2.2 FaaS Challenges

We list the challenges in a FaaS system as follows:

- The strict performance goals of FaaS workloads can potentially lead to poor system utilization, where nodes do not operate anywhere near their peak processing capability.
- "Backfilling" tasks to increase resource utilization is also not desirable at all times due to the detrimental effect it can have on the performance of the tasks.
- Since the runtimes for these functions is very small, any runtime system dynamically provisioning resources would have to be able to achieve very fine-grained telemetry, potentially on the micro-second scale. As evaluated in [9], computer systems of today significantly lack support for microsecond-scale events.

- Lack of a standardized set of FaaS workloads precludes the practicality of research in the area as well.

2.3 Motivation

Static and dynamic interference analysis and associated scheduling could potentially help mitigate the performance concerns. In that area, Dirigent[8] is a paper that this work borrows some ideas from. Composing of a profiler, predictor and controller, Dirigent dynamically partitions resources and throttles tasks based on a pre-determined profile of the application. Similarly, CPI2[10] identifies performance outliers using CPI information, determines the antagonist applications which are likely the perpetrators via online cross-correlation analysis and ameliorates the situation by throttling the latter. Resource pressure metrics (PSI[11]) for CPU, memory and I/O also exist but are very coarse-granular. For cloud-based workloads, the CPU Manager for Kubernetes [2, 12] does static allocation of resources with exclusivity. Figure 2.1 shows how the latency values and variability drop with the CPU Manager. All of the above exhibit the usability and potential advantages of static and dynamic techniques to provide better performance guarantees and as a result, better SLAs.

The transformation to FaaS has also attracted the attention of many inquisitive developers who have explored a fair bit of FaaS performance across languages on various popular FaaS service providers. Different experimental setups [13, 14, 15, 16, 17, 18] show variance in latencies across PL and also sometime across resource allocation. These information have suggested that hardware centric knowledge can be leveraged to normalize these experimental values. Also, smaller variance in latency can be beneficial for cloud providers to estimate resource utilization over

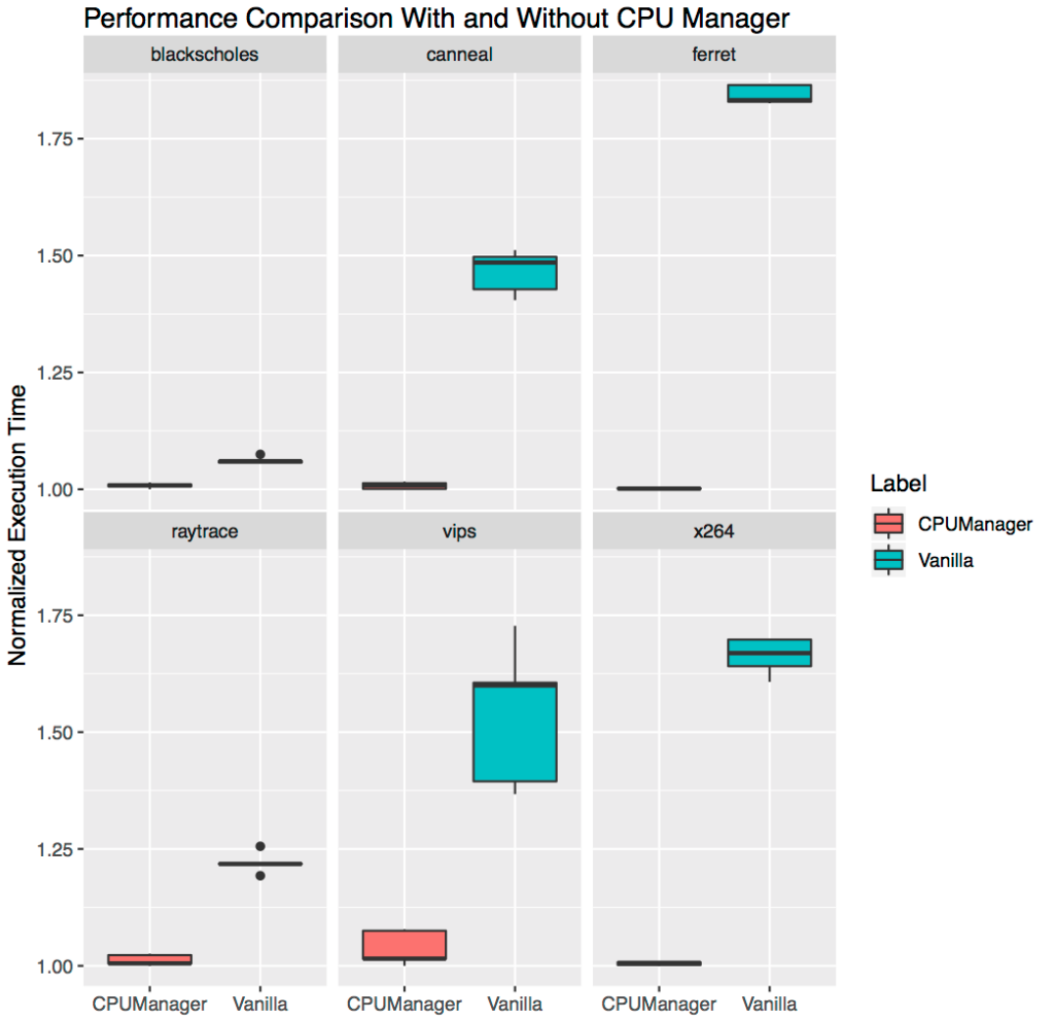

Figure 2.1: Performance effects of the CPU Manager for Kubernetes. Exhibits reduced latencies and latency variabilities from the Vanilla Kubernetes use case. [2]

a particular time-slice so as to be able to run appropriate tasks. These motivate us to research about the scope of hardware in understanding FaaS workloads and characterizing them for better scheduling or orchestration mechanism.

Chapter 3

Evaluation and Setup

We set up a local cluster with OpenFaaS to perform interference analysis. As mentioned earlier, this should give us more insights into the perspectives of a datacenter, how they can detect interference, do online analysis and perform QoS-aware scheduling. Both of the works below were in the context of micro-services but the ideas can easily be extended to functions, functions effectively being very lean and smaller micro-services.

3.1 Workloads

3.1.1 Microbenchmarks

Foreground Tasks

These tasks are small, user-facing and performance/latency-critical which are meant to be run via FaaS functions. Specifically, from the examples mentioned earlier, we run the following tasks:

- Face detection on a static image

- General matrix-matrix multiplication (building block of any neural network) on two arrays of size 200x200 each
- Phantomjs, a headless browser loading `www.cnn.com`

Background Tasks

These batch-jobs are long-running in the background which support streaming, online data analysis kind of services. These help with the efficient utilization of data-center resources. Specifically, we run the following programs tasks:

- Programs from the SPEC 2017 benchmark suite:
  - mcf (combinatorial optimization and scheduling)
  - cactuBSSN (solving Einstein equations in vacuum)
  - imagick (online image manipulation)
  - leela (AI-based Go playing engine)
- Grid search for parameter-space exploration for support vector machines

Since Docker Swarm does not provide us with hooks to taskset the containers and the processes running inside each to a specific core, we run the rate versions of the SPEC CPU2017benchmarks with reference inputs and number of copies equal to the number of cores. This is done so as to maximize resource utilization. Subsequently, the tasks are run with and without tasks to test if there is interference/contention.

3.1.2 Benchmark Applications

Due to the key features of functions being -

- stateless: persistent storage over network only

- loosely coupled tasks: can scale functions independently
- asynchronous: event-triggered
- latency-critical
- network bandwidth sensitive

we identify the following kinds of workloads/applications to be ideally suited for FaaS. We also detail the exact benchmarks implemented for each of the classes.

Data transformation

Face Detection (FD) This is a Pigo (Pixel Intensity Comparison-based Object detection) face detector implemented in Go. It is characterized by high processing speed and no pre-processing of data. Selected inputs are images, sizes of whose range from 30kB to 3MB and also varied number of concurrent requests. This is done so as to vary the computational and concurrent load to the application.

Big data applications

Sentiment Analysis (SA) This is a machine learning 'testing' application implemented in Python. Its characteristics are latency-critical and parallel processing of text statements. Input size varies from a few bytes to 32kB of JSON files which corresponds to analysis of simple tweets to long posts on social media. Load testing is also done by varied number of concurrent requests.

Machine Learning Hyperparameter Optimization (ML) This is a machine learning 'training' application implemented in Python. This app is characterized by embarrassingly parallel jobs to explore the design parameter space. Different calls to it are different optimization iterations or different accuracy required for the

model (100 parameters with 100 to 10k iterations). Load testing is again done by varied number of concurrent requests.

Web applications

Github Webhook + Sentiment Analysis (WH) This is implemented as a real web-app calling our Sentiment Analysis Python function from before. This is event-triggered (on a pull request, commit, issue, etc.) and does analysis of the comments. Inputs again vary from simple to complex comments with the input JSON file varying from a few hundreds of bytes to low kBs.

3.2 Baseline Related Work

We create static models of baseline related work for comparison. We do this because our own model is static and thus, we try to approximate the baseline related work into a static model as closely as possible. We outline both our baseline related work and similar-static models for the baseline related work in the subsections below.

3.2.1 CPI2

CPI2 [10] uses cycles-per-instruction data obtained by hardware performance counters to identify problem, select the likely perpetrators, and then optionally throttles them so that the victims can return to their expected behavior. It automatically learns normal and anomalous behavior by aggregating data from multiple tasks in the same job. CPI2 has been rolled out to all of Google's shared compute clusters. Specifically, CPI2 does the following:

- Observe the run-time performance of hundreds to thousands of tasks belonging to the same job and learn to distinguish normal performance from outliers.

- Identify performance interference within a few minutes by detecting such outliers.
- Determine which antagonist applications are the likely cause with an online correlation analysis and ameliorate by throttling or migrating the antagonists.

3.2.2 CPI2-Static

CPI2-Static is a static orchestration model that we create to closely resemble CPI2. Using CPI as a PMU metric has the following caveats according to CPI2.

- CPI is shown to well correlated with application-level behavior.
- Other techniques are required to detect and handle network and disk interference effects but there are enough examples of CPU interference to make that problem worth addressing.

CPI2 finds a positive stable correlation between changes in CPI and changes in compute-intensive application behavior. We compute the CPI distribution for all copies of the same task for all benchmark applications and all batch jobs. Rather than using a rank-order of a list of suspects based on heuristics like CPU usage and cache miss rate, CPI2 chooses to use the correlation values of the *victim's* CPI and the CPU usage of the *suspects*. To resemble this closely, we also statically correlate the CPU usage of batch workloads and CPI distribution of the benchmark applications to figure out which pairs of colocation will potentially *interfere* and which ones will not.

3.2.3 Bubble-up

The key insight of Bubble-up [19] is that predicting the performance interference of co-running applications can be decoupled into two steps:

- measuring how much an application suffers from different levels of pressure on the shared memory subsystem
- measuring the pressure on the memory subsystem an application generates

The two step methodology for Bubble-up is as follows.

- In step 1, the authors characterize the sensitivity of each application task to pressure in the memory subsystem. They use a carefully designed stress test called the *bubble* to iteratively increase the amount of pressure applied to the memory subsystem.
- In step 2, the authors characterize the *contentiousness* of each application task in terms of its pressure on the memory subsystem, something which they call the *bubble score*.

With the sensitivity curves and bubble scores of each application they are able to preisely predict the performance degradation from arbitrary colocations.

3.2.4 Bubble-up-Static

We try to approximate a static model to Bubble-up as closely as possible. Here, we use a time-series profile of page faults per kilo-instructions and cache MPKI to understand every application's pressure on the memory subsystem. Again, we do a correlation analysis between the respective time profiles of benchmark applications and batch workloads and use the correlation profile to colocate specific benchmark applications and batch workloads.

3.2.5 Dirigent

Dirigent [8] consists of three components.

- Offline execution profiler: Collects execution time and instruction count on each sampling granularity.
- Execution time predictor: Predicts application execution time by tracking actual progress, comparing this progress to the profiled data, and computing a *time penalty* experienced by the application and projecting it forward.
- Performance controller: Dirigent monitors the performance of FG applications online and uses the predictor to determine whether these applications are progressing faster or slower than necessary to meet their latency goals

Dirigent does not strive to minimize the execution time of latency-critical apps rather minimize the latency variation while meeting their latency targets. Thus, if a latency-critical app is predicted to finish early, it is deprioritized and vice versa. This prioritization/deprioritization is done by controlling the frequency at which each core operates, partitioning the last-level cache (LLC) and by pausing background jobs, whenever necessary.

3.2.6 Dirigent-Static

We do similar offline static profiling as Dirigent to come up with a time-series profile of instructions count executed in each sampling interval (which we keep the same as Dirigent, 5ms). Subsequently, to resemble the execution time predictor and performance controller closely, we also statically correlate the CPU usage and cache MPKI of batch workloads and the time-series profile of the benchmark applications. The CPU and memory usage of the batch workloads would help us determine the

interference of batch workloads with the benchmark applications. Based on this correlation, we can colocate benchmark applications and batch jobs not based on minimizing latency but on minimizing latency variance while still meeting QoS targets.

3.2.7 Heracles

Heracles [20, 21] argues that for workloads with performance measured by throughput, such as batch jobs, using IPC as a metric is a great way to directly measure how well an application is running. However the authors claim, because IPC is a measure of throughput, it cannot capture how well an user-facing latency-critical application is meeting its QoS target latency. They show using queuing theory how IPC cannot capture the latency behavior of an application.

Heracles uses four mechanisms to mitigate interference.

- Core isolation using Linux's cpuset cgroups to run latency-critical applications and batch workloads on different sets of cores.
- LLC isolation using Intel Cache Allocation Technology (CAT).
- Isolate DRAM bandwidth by periodically tracking bandwidth usage through performance counters.
- Power isolation using hardware features like CPU frequency monitors, RAPL and DVFS.

3.2.8 PARTIES

PARTIES [22] is a QoS-aware resource manager that enables an arbitrary number of interactive, latency-critical services to share a physical node without QoS violations.

Unlike CPI2 and Dirigent which focused on colocating latency-critical applications and batch workloads, PARTIES focuses on colocating multiple latency-critical applications with an aim to meet everyone's QoS targets.

3.3 Setup

3.3.1 Microbenchmarking

AWS Lambda is the most widely used FaaS provider on public cloud setups. We experimented with AWS Lambdas to characterize latency via cold start timings, network latency, interface latency and the actual integer and floating point compute latency. Via this approach, we tried to break down the end-to-end latency. The setup was AWS Lambda with varying memory limits (at a granularity of 64MB).

3.3.2 Non Load-testing Environment

Table 3.1 details our local cluster for the non load-testing environment consisting of two nodes (one physical and one virtual), one master and both as workers. A small hack was done to enable the master as one of the worker nodes as well. The orchestration is done using Kubernetes and the OpenFaaS CLI and GUI are setup on the cluster. Kubernetes is chosen as it is the most prevalent in industry although Docker Swarm was also experimented with. To replicate a deployment scenario, memory was constrained to a maximum of 512MB and the number of hyperthreaded cores to 1.

Three inputs were chosen from a representative set of more than 25 inputs of varying sizes. These three inputs can be considered to be sizes of *low*, *moderate* and

Architecture	x86_64
Model	Intel Core i7-4770K @ 3.50GHz
Cores	4
Threads per core	2
L1D, L1I, L2, L3 Cache	32K, 32K, 256K, 8192K

Table 3.1: Local Setup Configuration

high. We construct similar three inputs from a larger representative input set for all four benchmark applications and show results for only these three for brevity. A *non load-testing* environment implies that we experiment with individual non-concurrent request for each of the benchmark applications with each of the inputs. We do not do any concurrent requests with different inputs in this environment. Concurrent requests with a QoS latency target is a more realistic deployment scenario and we evaluate it in the next section.

3.3.3 Load-testing Environment

The setup is similar to the non load-testing environment (3.1). The difference is in how we run the experiments. Rather than individual requests with varying input sizes, we adopt a more realistic and deployment-related scenario. The two worker nodes run the functions invoked and we add a third node for the open-loop load generation. The load generation is done using wrapper scripts on top of the docker containers.

We create reasonable QoS target latencies for each of the benchmark applications and corresponding peak loads (maximum number of concurrent requests that can be made while still meeting the QoS target latency). We find out the performance slack (percentage of peak performance needed to meet the QoS goals) available at low to moderate loads. It is at these low to moderate loads when intu-

itively, we have considerable performance slack, where we try to colocate workloads intelligently.

Innately, this is what would happen in the real world and meeting the QoS target while maximizing hardware utilization is our goal.

Chapter 4

Results

4.1 Microbenchmarking

4.1.1 API Latency Characterization

Lambda supports CLI interface and provides an API Layer to invoke functions. We used this feature to characterize network latency to AWS servers by running batched and individual API calls from a remote machine and from an EC2-instance deployed on a server in the same region as the function was being deployed. We invoke 100 requests with a concurrency of 10 requests to see scaling features of AWS. Figure 4.1 shows how the network latency (comprised to the DNS lookup, TLS handshake and the actual network propagation latency) affects total response latency. As we can notice, the initial requests take significantly more time than the subsequent queued jobs. This is because of the nature the FaaS stack has been built, which is vastly different from VMs or EC2-instances. The stack spins up containers which are pre-packaged with our functionality when a request is seen by the API layer. Once the request has been serviced the container, is killed and cleaned up. For performance,

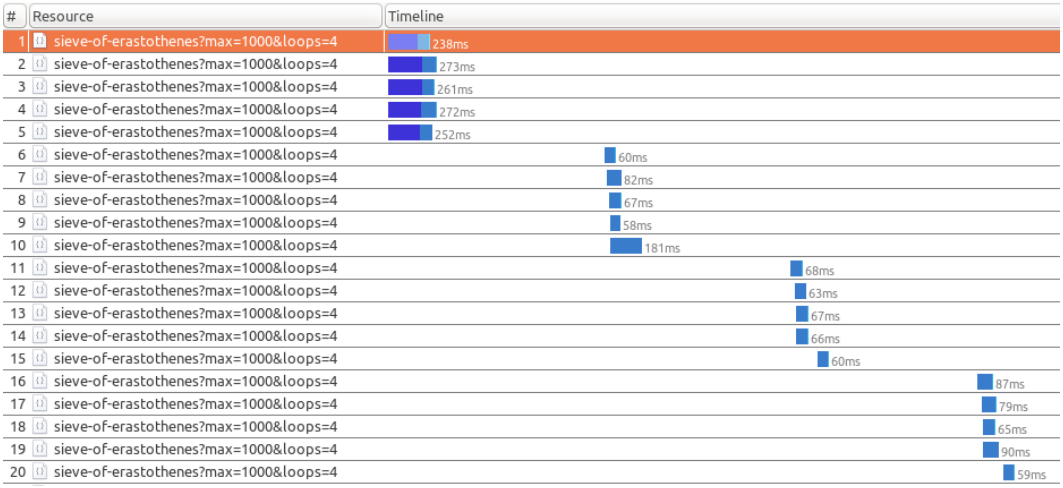

Figure 4.1: First runs after a few minutes exhibits longer runtimes: *cold-starts*.

AWS, and other FaaS stacks, keep the container alive for a duration of 15 minutes to half an hour before it is cleaned. The first request has to wait for the orchestrator to set up the container and then the request is serviced while the queued requests later already have resources readily available and hence have shorted round-trip time. This initiation is termed *cold-starts*which pose as a significant roadblock to better SLA guarantees. For periodic cron-jobs or work-hour spikes in traffic in the worst case, this might lead to *cold-starts*on all request services and an *intelligent* scheduler should ideally predict usage and spin up containers at the right time to service these requests.

The total latency for any request comprises of *cold-starts*, network latency (DNS lookup and TLS handshakes), interface latency and actual compute of the function.

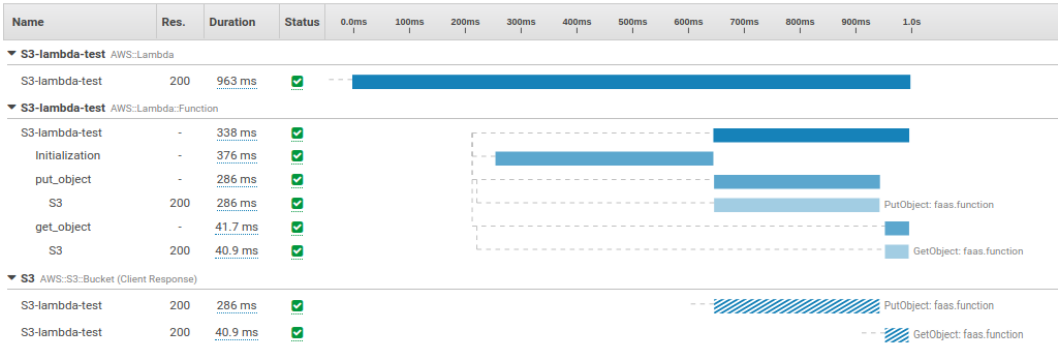

Figure 4.2: AWS X-Ray feature for accurate timings

4.1.2 Container Interface Latency

Functions usually are event-driven and the event is received as a JSON to an event handler. We have run experiments on Python and blogs suggest that latencies for different languages can vary. AWS provides APIs for AWS X-Ray which allows us to put markers in functions to get timings of user-defined regions as shown in Figure 4.2. We configured our workloads with X-Ray to study how does the overall function latency change based on payload of the function. Placing a marker right at the beginning of the python program allows us to monitor how long it takes for the container to receive the input payload after the function has been invoked which would be dependant on the network latency and the bandwidth. We varied the payload sizes from 2 Bytes up to 4MB to visualize the latency increase. Figure 4.3 shows the latency distribution for the same. The AWS lambda in this case is essentially doing nothing as we are only benchmarking the interface latency as a function of payload size. We observe that the interface latency starts increasing after 16kB of payload and the latency distribution also shows the presence of outliers even for the smallest payload which can have an adverse effect on the SLAs. These outliers show us that interconnects are contended for in some cases.

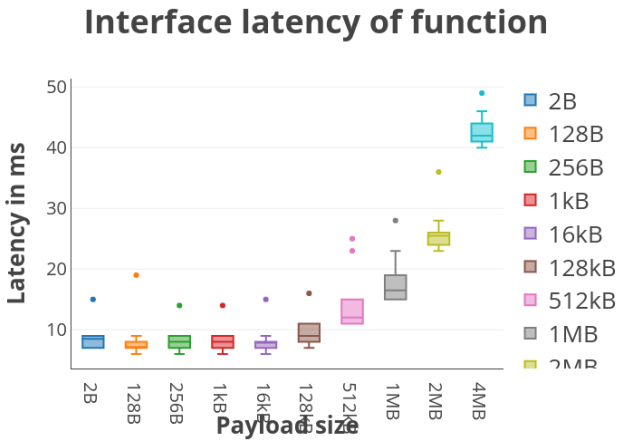

Figure 4.3: Container interface latency shows a sudden increase after 16KB of payload size

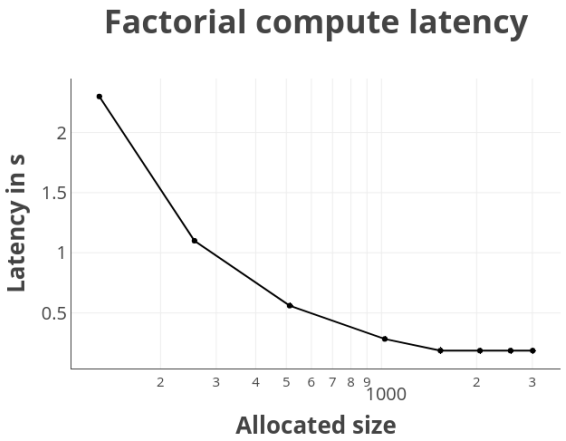

Figure 4.4: Latencies for Factorial decrease almost linearly as we allocate more memory to the function.

4.1.3 Performance Impact

Docker containers, which are the most widely used container service, allows container resource limitations on CPU and memory for a container. AWS Lambda ties these two tightly by allowing only modification of memory. Allocation of higher memory for a function automatically assigns higher CPU resources, probably in terms of

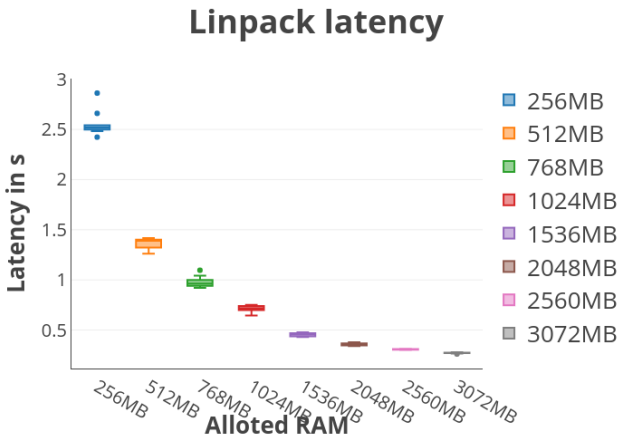

Figure 4.5: Latencies for Linpack decrease almost linearly with increase of allotted memory

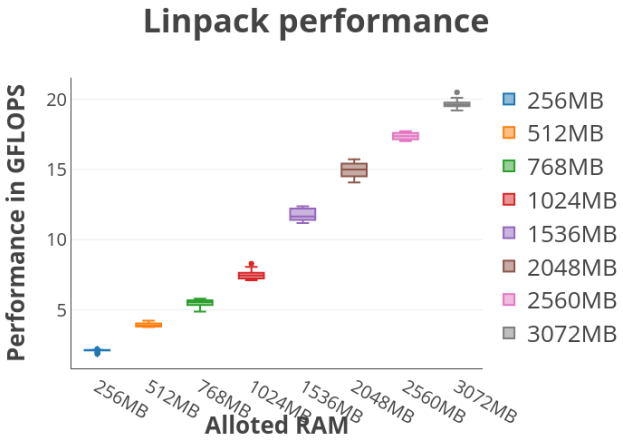

Figure 4.6: Performance of Linpack accordingly increases with increase of allotted memory

time slices or cgroups. Since the resource limitations are tightly bound, we perform integer and floating-point compute intensive workloads on Lambda by varying the allocated memory to the function. We use the Linpack benchmark for floating points ops and factorial calculation to stress integer operations. We choose factorial since it is a smaller kernel and earlier blogs have used the same for benchmarking the integer

compute units. We isolate this compute latency via AWS X-ray APIs and observe that the latencies for Linpack are much more variable than factorial. Figures 4.4 and 4.15 show the Linpack latency and performance (in GFLOPS) distribution and figure 4.6 shows the variation of Factorial latency graph with the allotted memory. The latter is just a single line since the computation is very deterministic.

All of the above characterization helps us gain some insights into how AWS schedules lambdas. However, AWS Lambda is a black-box to us in the sense that it does not provide users with very fine-grained hardware characteristics and reverse-engineering its scheduling policy only goes so far. Overall, it does not provide us with a suitable setup to conduct in-depth research. This prompted us to move to open-source FaaS projects for our study. A lot of such projects exist but a majority of them are in their initial stages and are difficult to work with (unstable codebases and/or buggy). We delve into some of the explored projects and our evaluations on those in the next section.

4.2 Benchmark Applications Characterization

4.2.1 High-level Metrics

Figures 5.7, 5.8 and 5.9 show different high-level metrics associated with each of the benchmark applications with varying input loads.

4.2.2 PMU Characterization

The following PMU metrics were used for characterization of the benchmark applications.

- IPC: instructions/cycle, **compute intensity**

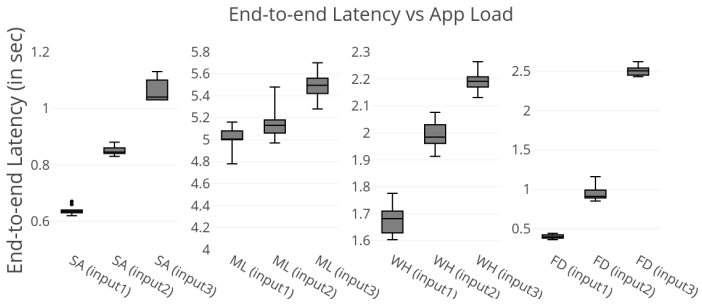

Figure 4.7: Latencies of benchmark applications ranges from few hundred milliseconds to a few seconds

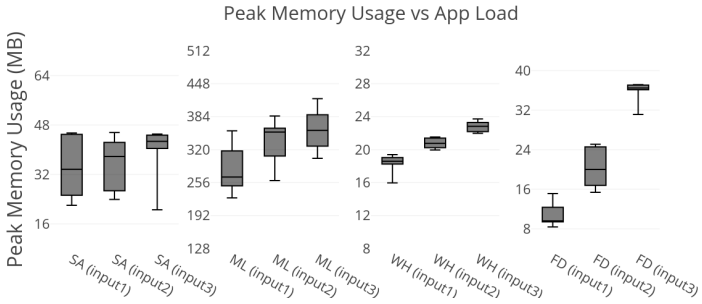

Figure 4.8: Memory working set for benchmark applications varies from tens to hundreds of MBs

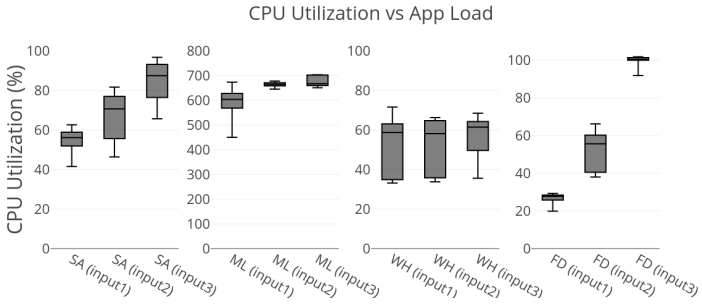

Figure 4.9: CPU Utilization of benchmark applications

- Cache MPKI: misses/kilo-instructions, **memory pressure**
- Branch MPKI: mispredicts/kilo-instructions, **pipeline front-end stalls**

- Page faults: long latency memory trips

The challenges in collecting the same was that we did not have any core-level visibility/monitoring on OpenFaaS/AWS Lambda. Thus, we extracted benchmark applications out of Kubernetes+OpenFaaS and taskset the native containers and collected performance counters for the particular core. Figures below show cache MPKI, IPC and page faults with increasing app load.

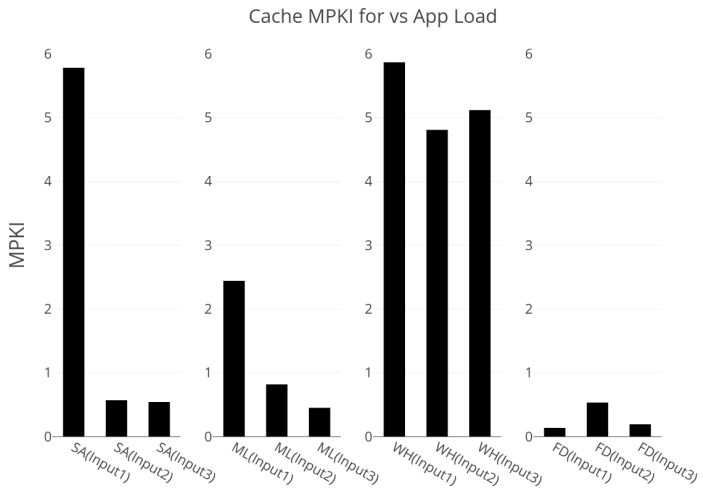

Figure 4.10: Cache MPKI with increasing app load

4.3 Non Load-testing Environment

4.3.1 Colocated Setting

In the non load-testing environment, we compare the colocation by Kubernetes and on a intelligent static PMU-based performance model. The latter uses -

- IPC: <1 usually memory bound and > 1 usually compute bound
- Cache MPKI: Quantifies the *intrusiveness* of an app

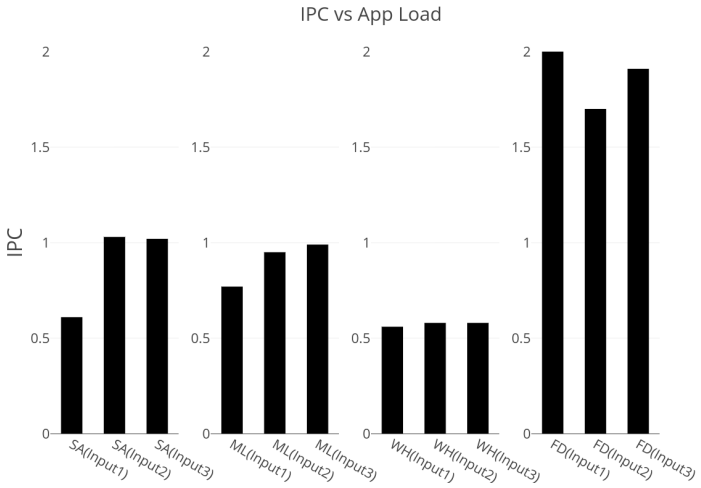

Figure 4.11: IPC with increasing app load

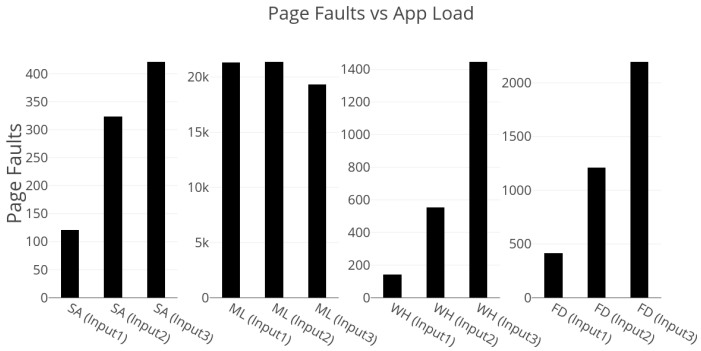

Figure 4.12: Page faults with increasing app load

- Page faults: Small latency FaaS workloads should generally not incur too many page faults

The figures below show colocation with Kubernetes as the orchestrator and our intelligent static PMU-based performance model. Clearly, in the case of webhook (WH), the latter performs better than Kubernetes. In the same case, colocating intelligently based on PMU characterization yields 73% less variance in latency.

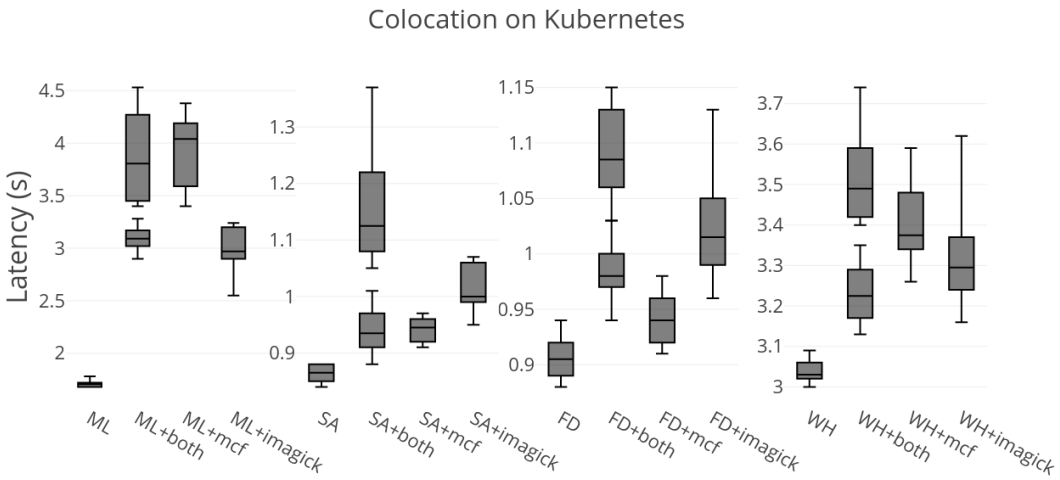

Figure 4.13: Colocation on Kubernetes

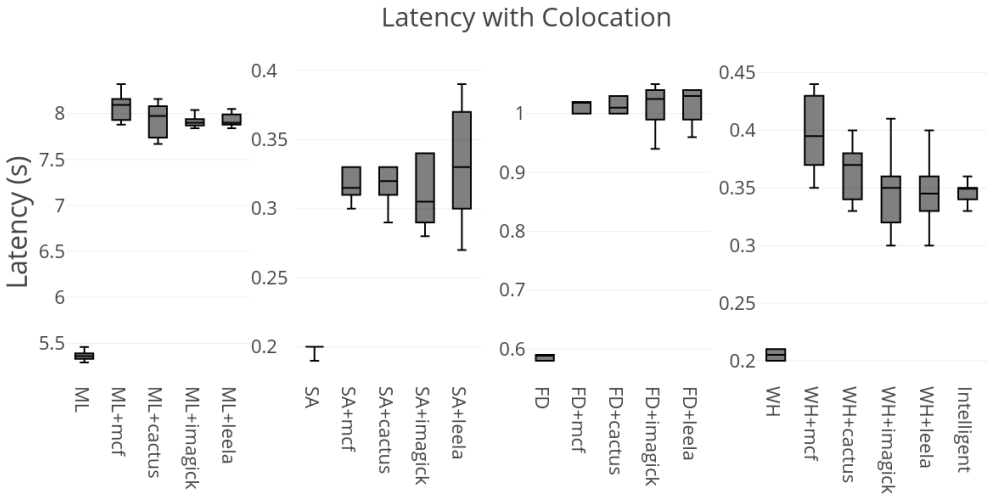

Figure 4.14: Intelligent static colocation. In the right-most case of the webhook application, colocating intelligently based on PMU characterization yields 73% less variance in latency.

4.4 Load-testing Environment

Datacenters need to move to a colocation of latency-sensitive workloads and batch jobs to maximize resource utilization. Significant performance slack exists (low hardware utilization) when servers are not running at max load. This problem is aggravated for FaaS workloads because of their event-triggered (asynchronous) and ephemeral nature. Colocate batch workloads intelligently to use inherent performance slack at low to moderate loads for latency-critical FaaS workloads.

4.4.1 Load Generation

This kind of environment is a more realistic one for deployment/production. We create reasonable peak loads for each benchmark app and estimate the QoS target - the latency which should not be crossed in any case. This we call as the *QoS latency* which corresponds to the *peak load*. We summarize how we load-test for different apps as below:

- Machine learning hyperparameter optimization (ML)
  - Multiple concurrent hyperparameter optimization requests
  - Each request trying to do the optimization for a random number of iterations
- Face detection (FD)
  - Multiple concurrent face detection request on random images
- Sentiment Analysis (SA) / Webhook (WH)
  - Concurrent calls for sentiment analysis / webhook for different texts

4.4.2 Performance Slack

We quantify latencies for all benchmark applications as a function of percentage of max load. We only show the latencies for ML for brevity below.

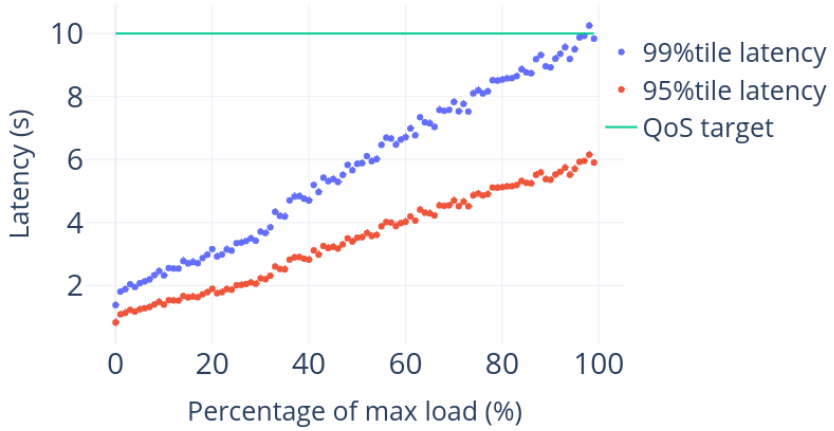

Figure 4.15: ML latency at loads varying from 0 to 100% of maximum load. Maximum load corresponds to the QoS latency guaranteed by the cloud service provider.

Based on the ML latencies at various loads above, we quantify *performance slack* available for each benchmark application as a function of percentage of max load. *Performance slack* is the percentage of peak performance needed to meet the QoS target latency. Figure 4.16 shows performance slacks for all benchmark applications at all loads.

4.4.3 Colocated Setting

Now, we colocate all benchmark applications with all batch jobs on a Kubernetes setup. The percentage slowdowns are averaged out across all loads from 0 to peak load. In general, we see batch jobs incurring more slowdown than benchmark ap-

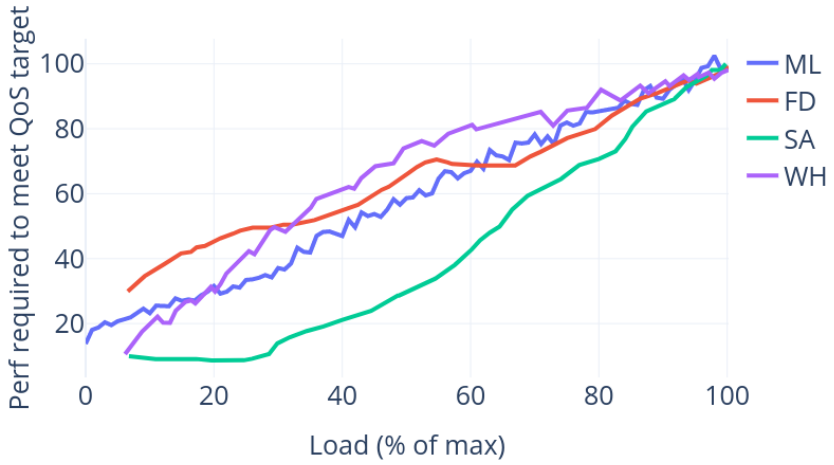

Figure 4.16: Performance slack available at lower loads.

plications.

Subsequently, we colocate for all possible loads based on our static performance counter based model. In general, we see a lower slowdown with a lower variance in latency. Both of the graphs are shown side by side as below.

Figure 4.17 shows the speedup for all benchmark applications and batch jobs over the Kubernetes setup. On average, benchmark applications show 9% speedup whereas batch jobs 19% over the Kubernetes baseline.

Figure 4.18 shows the average performance slowdowns for the CPI2-Static, Bubble-up-Static and Dirigent-Static.

Now, we will move on to summarizing all of the results for all of the related work (CPI2-Static, Bubble-up-Static and Dirigent-Static), the Kubernetes baseline and our intelligent PMU-based model. Figure 4.19 compares the latency speedup for the related work and our model on top of the Kubernetes orchestration baseline. We observe that we perform better than CPI2-Static but worse than Bubble-up-Static

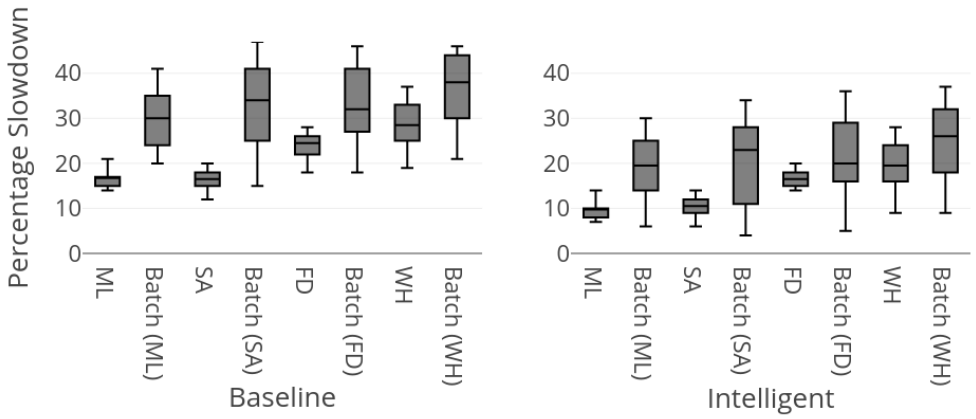

Figure 4.17: Comparison of average performance slowdown on colocation with Kubernetes and our intelligent static performance model

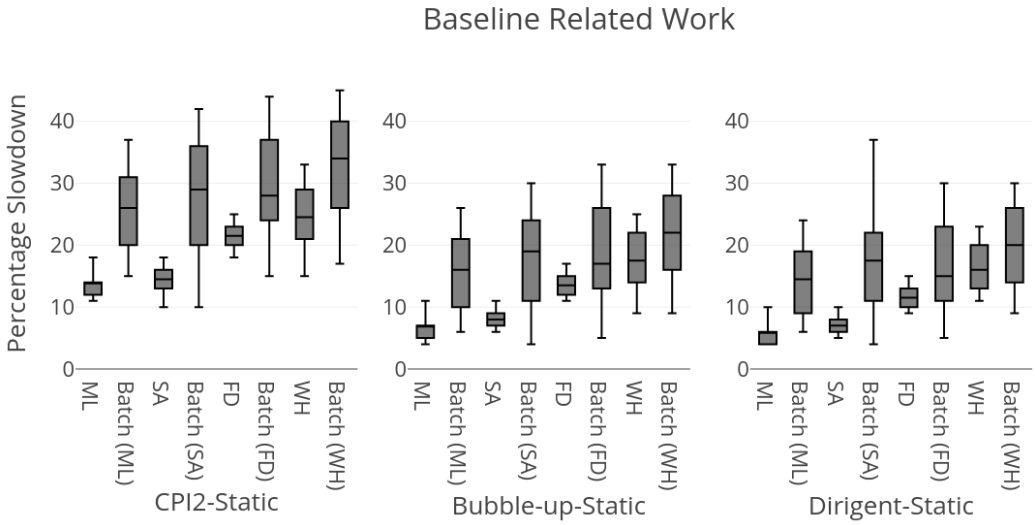

Figure 4.18: Average performance slowdown for the baseline related work - CPI2-Static, Bubble-up-Static and Dirigent-Static

and Dirigent-Static. Our model outperforms CPI2-Static by 6.8% for benchmark applications and 9% for batch workloads. However, it is outperformed by Bubble-up-Static by 10.1% for benchmark applications and 11.8% for batch workloads and by Dirigent-Static by 22.9% for benchmark applications and 24.3% for batch workloads. This comparison is in the latency department and later, we move on to comparing the variance in latency.

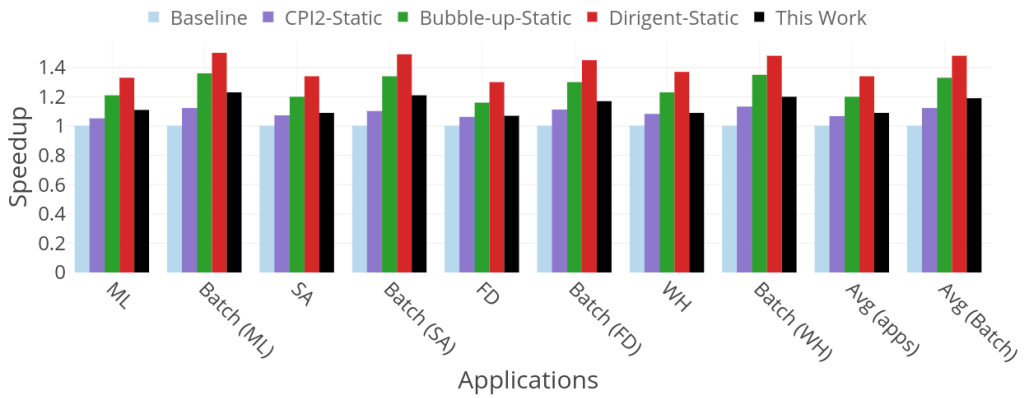

Figure 4.19: Latency speedup comparison for all baseline related work and our intelligent model with respect to a baseline Kubernetes colocation (higher is better)

Figure 4.20 compares the variance in latency for the related work and our model on top of the Kubernetes orchestration baseline. We observe that we again perform better than CPI2-Static but worse than Bubble-up-Static and Dirigent-Static. Our model outperforms CPI2-Static by 8.9% for benchmark applications and 9.6% for batch workloads. However, it is outperformed by Bubble-up-Static by 16.4% for benchmark applications and 12.5% for batch workloads and by Dirigent-Static by 30.1% for benchmark applications and 27.3% for batch workloads.

As we can see, our model performs worse than Bubble-up-Static and Dirigent-Static. Iterating on our model, we include one more FaaS feature -

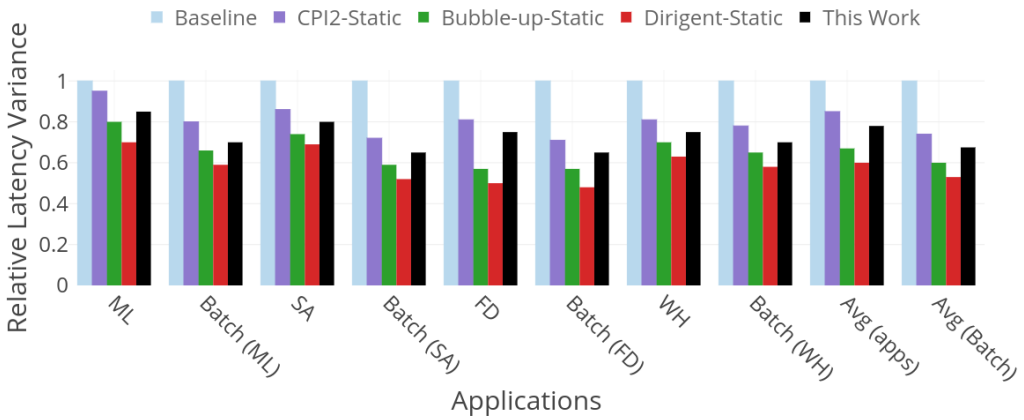

Figure 4.20: Latency variance comparison for all baseline related work and our intelligent model with respect to a baseline Kubernetes colocation (lower is better)

- Time-series profiling of CPU usage for batch workloads and correlating the same with time-series profile of benchmark application progress (execution time, instructions executed): We include this to get a sense of the sensitivity of benchmark application progress to the compute intensive nature of the batch workloads.

Adding this feature to determine colocation, we run our model again and this time around it colocates all of the benchmark applications with the same background batch jobs as Dirigent-Static except for the webhook (WH) application. WH is colocated with mcf+cactuBSSN by our model (same as Bubble-up-Static) whereas it is colocated with mcf+leela by Dirigent-Static.

Figures 4.21 and 4.22 are the latency speedup and relative latency variance graphs for our augmented model (with the new FaaS feature) with the same baseline Kubernetes and static versions of baseline related work models. Now, our model outperforms Bubble-up-Static by 8.8% for benchmark applications and by 9.0% for batch workloads. Owing to the worse performance for WH, Dirigent-Static still

outperforms us on average - by 2.7% for benchmark applications and by 2.1% for batch workloads.

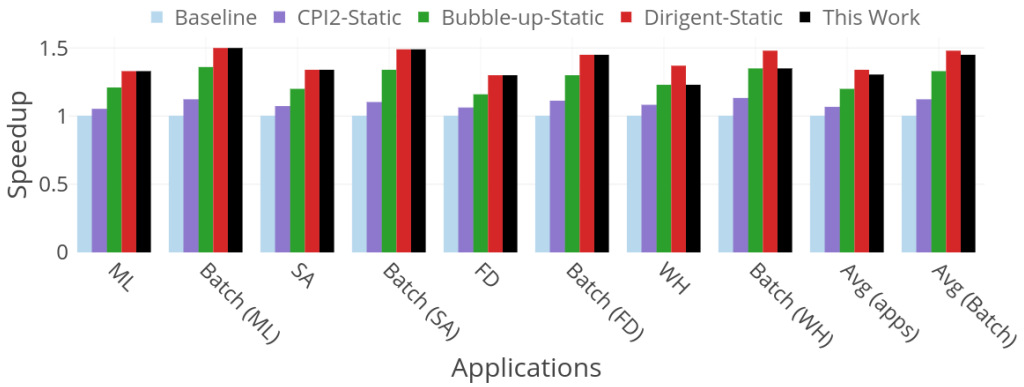

Figure 4.21: Latency speedup comparison for all baseline related work and our augmented intelligent model with respect to a baseline Kubernetes colocation (higher is better)

For relative latency variance, our model outperforms Bubble-up-Static by 8.1% for benchmark applications and by 9.1% for batch workloads. Again, owing to WH, Dirigent-Static outperforms us on average in this department too - by 3.3% for benchmark applications and by 3.8% for batch workloads.

This augmented model of ours performs worse than Dirigent-Static for WH because Dirigent-Static also correlates a time-series profile of cache MPKI for the batch workload. Thus, leela having a smaller cache MPKI envelope than cactuBSSN is chosen by Dirigent-Static to colocate with the memory intensive sentiment analysis part of WH. This is not captured by the overall scalar cache MPKI value because cactuBSSN is a long running program with high cache miss spikes but overall lower MPKI.

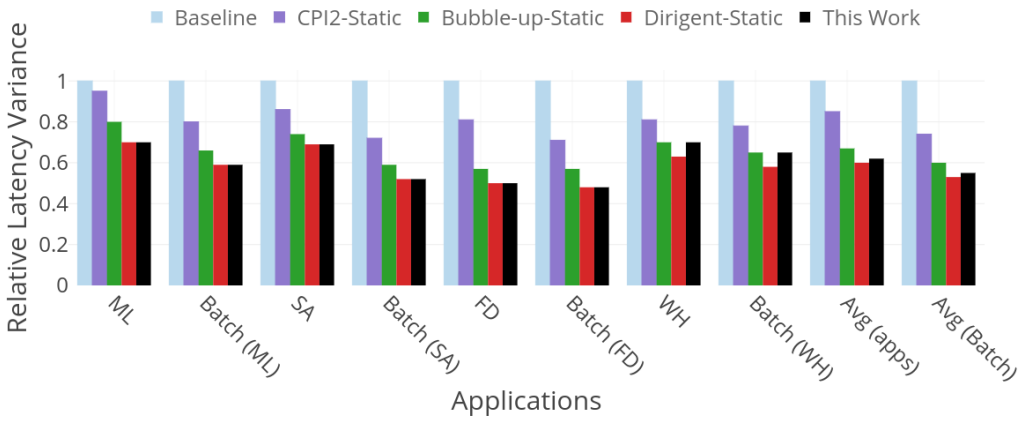

Figure 4.22: Latency variance comparison for all baseline related work and our augmented intelligent model with respect to a baseline Kubernetes colocation (lower is better)

Chapter 5

Related Work

Since the advent of the FaaS programming model in early 2014, there has been little research into this from the systems hardware industry. This is because of the closed-source opaque model of current FaaS providers provided by cloud vendors like Amazon Web Services Lambda, Microsoft Azure, Google Cloud Functions. Oracle Fn and IBM Apache OpenWhisk are the only open-source ones. This related work is divided into related work into AWS Lambda, benchmarking of FaaS platforms and QoS for datacenters.

5.1 AWS Lambda

Although Amazon has been secretive, Lambda being the most popular functions platform, a lot of research covers some deep dive towards Lambda architecture. [23] have compared all function platforms from the top cloud vendors. They have evaluated cold-starts and a few compute metrics but lack a set of benchmarks. Apart from reseach, AWS has published whitepapers [24, 25] describing their software architecture and usability of Lambda suites, which is informative for an understanding of the

stack. Our benchmarking efforts extend that work on a standard set of workloads.

5.2 FaaS

Since function-as-a-service paradigm is relatively new, micro-benchmark and benchmarking application to characterize FaaS is not yet well explored. There have been related research into micro-services [26] bench-marking and few initiatives from companies like IBM[27] to popularize benchmark applications which can be standardized. Recently, Sriraman et. al. [28] came up with a suite of micro bench-mark for FaaS applications but we are still a long way from an actual benchmark workload suite. Tail latency specific benchmarks [29] provide a good starting point but a lot of other characteristics are necessary to reach a good set.

5.3 QoS of Datacenters

Quality of Service in datacenters is a widely studied topic [30, 8, 22, 31, 32, 33]. There have been significant work done by both hardware and software researchers towards optimization of data-centers resources. We pick most of our implementation methodology from Dirigient [8] and Stretch [34]. Tail latency [35, 36, 37, 38, 39, 40] is yet another domain of vast research for workloads on cloud servers. A long-tailed or heavy-tailed probability distribution is one that assigns relatively high probabilities to regions far from the mean or median[41]. Most of the work is done for co-location and power optimization for Virtual Machines since they are long running and low utilization instances. But this project extends these concepts to track and optimize resource utilization at milliseconds levels. As far as we know, no such work has been done for FaaS workloads.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

Even with a simple PMU-based static performance model for orchestration, we achieve a 9% speedup and a 19% reduction in latency variance for our FaaS benchmark applications over a traditional Kubernetes orchestration. Compared to baseline related work i.e. static versions of CPI2, Bubble-up and Dirigent, our static model performs better than CPI2 but worse than Bubble-up and Dirigent both in terms of latency and variance in latency.

Our work is able to track *memory-boundedness* on top of compute and thus, it performs better than CPI2-Static. Additionally, Heracles [20, 21] shows using queuing theory that CPI does not capture the latency behavior of an application. Bubble-up-static performs slightly better since they use an offline time-series *memory pressure* profile rather than static values as our work. Dirigent-Static performs the best since it uses correlation analysis of time-series profiles of latency-critical application progress and memory and compute-boundedness of batch workloads.

In summary, datacenters need to move to a colocation of latency-sensitive

workloads and batch jobs to maximize resource utilization. Significant performance slack exists (low hardware utilization) when servers are not running at max load. This problem is aggravated for FaaS workloads because of their event-triggered (asynchronous) and ephemeral nature. Colocating batch workloads intelligently to use inherent performance slack at low to moderate loads for latency-critical FaaS workloads can result in better hardware utilization while still meeting the QoS target latencies.

6.1.1 Feature Vectors for FaaS Orchestration

Feature vectors to be used for intelligent FaaS orchestration seem to be roughly similar to a microservices orchestrator. A static version of Dirigent (which works for general user-facing latency-sensitive and batch workloads) seems to be working well and our model comes close to its performance (only WH performs worse because of the reason mentioned earlier). In general, the following features together seem to be working well for an intelligent static FaaS orchestration -

- Overall IPC
- Overall Cache MPKI
- Overall Page faults per kilo-instructions
- Offline correlation analysis of CPU usage profile for batch workloads and benchmark application progress profile.

6.2 Future Work

6.2.1 Heterogeneous Orchestration

Orchestration on heterogeneous nodes and across nodes using Kubernetes augmented with a runtime performance model of applications is left for future work. Such an orchestrator would be aware of the function-graph and background/throughput tasks and colocate in-flight/online. Migrating workloads between different kinds of nodes (such as in ARM big-Little) should also help the performance of certain workloads. It would also be preferable if the scheduler does not require offline profiling as that would be cumbersome and difficult to achieve in some cases.

6.2.2 Opportunities for New Hardware

Simpler and often-used functions can be configured on in-line FPGAs, for example. This would help in reducing the overhead for executing those particular functions and would help the performance of any micro-service using the same functions.

Bibliography

[1] "apache/incubator-openwhisk: Apache openwhisk is a serverless event-based programming service and an apache incubator project.." https://github.com/apache/incubator-openwhisk.

[2] "Feature highlight: Cpu manager - kubernetes." https://kubernetes.io/blog/2018/07/24/feature-highlight-cpu-manager/.

[3] "What is serverless computing? — serverless definition." https://www.cloudflare.com/learning/serverless/what-is-serverless/.

[4] "Aws lambda - serverless compute - amazon web services." https://aws.amazon.com/lambda/.

[5] "Run swarm and kubernetes interchangeably — docker." https://www.docker.com/products/orchestration.

[6] "kubernetes/kubernetes: Production-grade container scheduling and management." https://github.com/kubernetes/kubernetes.

[7] "Evolution of serverless: Monolithic-microservices-faas." https://dev.to/jignesh_simform/evolution-of-serverless-monolithic-microservices-faas-3hdp.

[8] H. Zhu and M. Erez, "Dirigent: Enforcing qos for latency-critical tasks on shared multicore systems," in *the Proceedings of ASPLOS*, (Atlanta, GA), pp. 33–47, April 2016.

[9] L. Barroso, M. Marty, D. Patterson, and P. Ranganathan, "Attack of the killer microseconds," *Communications of the ACM*, vol. 60, no. 4, pp. 48–54, 2017.

[10] X. Zhang, E. Tune, R. Hagmann, R. Jnagal, V. Gokhale, and J. Wilkes, "Cpi 2: Cpu performance isolation for shared compute clusters," in *Proceedings of the 8th ACM European Conference on Computer Systems*, pp. 379–391, ACM, 2013.

[11] "psi: pressure stall information for cpu, memory, and io v2 [lwn.net]." `https://lwn.net/Articles/759658/`.

[12] "intel/cpu-manager-for-kubernetes: Kubernetes core manager for nfv workloads." `https://github.com/intel/CPU-Manager-for-Kubernetes`.

[13] "Aws lambda + serverless framework + python - a step by step tutorial - part 1 "hello world"." `https://hackernoon.com/aws-lambda-serverless-framework-python-part-1-a-step-by-step-hello-world-4182202ab`

[14] "Lambda internals: Exploring aws lambda - hacker noon." `https://hackernoon.com/lambda-internals-exploring-aws-lambda-462f05f74076`.

[15] "An introduction to serverless and faas (functions as a service)." `https://medium.com/@BoweiHan/an-introduction-to-serverless-and-faas-functions-as-a-service-fb5cec0417b2`.

[16] "Serverless architectures." https://martinfowler.com/articles/serverless.html.

[17] "Docker + servless = faas (functions as a service) — luke angel." http://lukeangel.co/cross-platform/docker-servless-faas-functions-as-a-service/.

[18] "What is function-as-a-service? serverless architectures are here!." https://stackify.com/function-as-a-service-serverless-architecture/.

[19] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: Increasing utilization in modern warehouse scale computers via sensible co-locations," in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-44, (New York, NY, USA), pp. 248–259, ACM, 2011.

[20] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: improving resource efficiency at scale," in *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, pp. 450–462, 2015.

[21] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Improving resource efficiency at scale with heracles," *ACM Trans. Comput. Syst.*, vol. 34, no. 2, pp. 6:1–6:33, 2016.

[22] S. Chen, C. Delimitrou, and J. F. Martinez, "PARTIES: QoS-Aware Resource Partitioning for Multiple Interactive Services," in *Proceedings of the Twenty Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, April 2019.

[23] M. Malawski, K. Figiela, A. Gajek, and A. Zima, "Benchmarking heterogeneous

cloud functions," in *Euro-Par 2017: Parallel Processing Workshops* (D. B. Heras, L. Bougé, G. Mencagli, E. Jeannot, R. Sakellariou, R. M. Badia, J. G. Barbosa, L. Ricci, S. L. Scott, S. Lankes, and J. Weidendorfer, eds.), (Cham), pp. 415–426, Springer International Publishing, 2018.

[24] "Aws-serverless-applications-lens.pdf." `https://d1.awsstatic.com/whitepapers/architecture/AWS-Serverless-Applications-Lens.pdf`.

[25] "serverless-architectures-with-aws-lambda.pdf." `https://d1.awsstatic.com/whitepapers/serverless-architectures-with-aws-lambda.pdf`.

[26] C. Delimitrou and C. Kozyrakis, "ibench: Quantifying interference for datacenter applications," in *2013 IEEE international symposium on workload characterization (IISWC)*, pp. 23–33, IEEE, 2013.

[27] "Benchmarking serverless: Ibm scientists devise a test suite to quantify performance - the new stack." `https://thenewstack.io/ibm-scientists-set-quantify-serverless-performance/`.

[28] "Iiswc2018-usuite-preprint.pdf." `http://akshithasriraman.eecs.umich.edu/pubs/IISWC2018-%CE%BCSuite-preprint.pdf`.

[29] H. Kasture and D. Sanchez, "Tailbench: A benchmark suite and evaluation methodology for latency-critical applications," in *Workload Characterization (IISWC), 2016 IEEE International Symposium on*, pp. 1–10, IEEE, 2016.

[30] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: modeling techniques and their applications," *Journal of Internet Services and Applications*, vol. 5, no. 1, p. 11, 2014.

[31] N. Kulkarni, F. Qi, and C. Delimitrou, "Pliant: Leveraging Approximation to Improve Datacenter Resource Efficiency," in *Proceedings of the 25th IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, February 2019.

[32] A. Beloglazov and R. Buyya, "Managing overloaded hosts for dynamic consolidation of virtual machines in cloud data centers under quality of service constraints," *IEEE Transactions on Parallel and Distributed Systems*, vol. 24, no. 7, pp. 1366–1379, 2013.

[33] S. Ranjan, J. Rolia, H. Fu, and E. Knightly, "Qos-driven server migration for internet data centers," in *Quality of Service, 2002. Tenth IEEE International Workshop on*, pp. 3–12, IEEE, 2002.

[34] A. Margaritov, S. Gupta, R. Gonzalez-Alberquilla, and B. Grot, "Stretch: Balancing qos and throughput for colocated server workloads on smt cores," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 15–27, Feb 2019.

[35] C. Delimitrou and C. Kozyrakis, "Amdahl's Law for Tail Latency," in *Communications of the ACM (CACM)*, August 2018.

[36] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martínez, "Workload characterization of interactive cloud services on big and small server platforms," in *Workload Characterization (IISWC), 2017 IEEE International Symposium on*, pp. 125–134, IEEE, 2017.

[37] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in *Proceedings of the ACM Symposium on Cloud Computing*, pp. 1–14, ACM, 2014.

[38] M. E. Haque, Y. He, S. Elnikety, R. Bianchini, K. S. McKinley, *et al.*, "Few-to-many: Incremental parallelism for reducing tail latency in interactive services," in *ACM SIGPLAN Notices*, vol. 50, pp. 161–175, ACM, 2015.

[39] Y. Zhang, D. Meisner, J. Mars, and L. Tang, "Treadmill: Attributing the source of tail latency through precise load testing and statistical inference," in *ACM SIGARCH Computer Architecture News*, vol. 44, pp. 456–468, IEEE Press, 2016.

[40] J. Dean and L. A. Barroso, "The tail at scale," *Communications of the ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[41] "Long-tail traffic - wikipedia." `https://en.wikipedia.org/wiki/Long-tail_traffic`.